1. Tempo disponibile 120 minuti.
2. Non è possibile consultare appunti, slide, libri, persone, siti web, ecc.
3. Scrivere in modo leggibile, su ogni foglio, nome, cognome e numero di matricola.
4. Le soluzioni agli esercizi che richiedono di progettare un algoritmo devono:
   - spiegare a parole l'algoritmo (se utile, anche con l'aiuto di esempi o disegni),
   - fornire e commentare lo pseudo-codice (indicando il significato delle variabili),
   - calcolare la complessità (con tutti i passaggi matematici necessari),
   - se l'esercizio ammette più soluzioni, a soluzioni computazionalmente più efficienti e/o concettualmente più semplici sono assegnati punteggi maggiori.

1. Calcolare la complessità $T(n)$ del seguente algoritmo `mystery1`:

```
algoritmo mystery1(n: Int) --> Int
    Int i = 1, tot = 0
    while (i <= n)
        i = i*2
        tot = tot + mystery2(n)
    endwhile
    return tot

algoritmo mystery2(m: Int) --> Int
    if (m == 1)
        return 77
    else
        Int v = 1
        for i = 1..m do v = v*2
        return 2 * mystery2(m-1)
  endif
```

Soluzione L'algoritmo `mystery1` utilizza `mystery2`. Iniziamo quindi l'analisi da tale secondo algoritmo, che risulta essere un ricorsivo con complessità $T'(m)$ caratterizzata dalla seguente equazione di ricorrenza:

$$T'(m) = \begin{cases} 1 & \text{se } m \leq 1 \\ T'(m-1) + m & \text{altrimenti} \end{cases}$$

vista la presenza di un for che nel caso in cui m sia maggiore di 1 esegue m cicli composti da operazioni con costo costante. Si noti che tale equazione non si basa su partizionamenti bilanciati e quindi non è possibile l'utilizzo del Master Theorem. Procediamo con il metodo dell'iterazione:

$$T'(m) = T'(m-1) + m = T'(m-2) + (m-1) + m = \cdots = \sum_{1=1\ldots m} i = \frac{(m+1) \times m}{2} = O(m^2)$$

Analizziamo ora `mystery1`. Tutte le operazioni hanno costo costante ad esclusione della invocazione a `mystery2` che si trova all'interno di un while che viene eseguito $O(\log n)$ volte, in quanto l'indice del ciclo inizialmente vale 1 e viene raddoppiato ad ogni ciclo fino a superare il valore n. Quindi vengono fatte $O(\log n)$ invocazioni a `mystery2`, ogni volta passando n come parametro; per quanto detto sopra ogni invocazione ha costo $O(n^2)$. Complessivamente il tempo di calcolo di `mystery1` è quindi $T(n) = O(n^2 \log n)$.

2. Dato un *albero23* inizialmente vuoto, effettuare le seguenti operazioni in ordine e mostrare lo stato dell'albero dopo ogni operazione: INSERT(15); INSERT(25); INSERT(2); INSERT(40); INSERT(20); DELETE(2); INSERT(90), INSERT(80); DELETE(15). Commentare brevemente le singole operazioni.

Soluzione Vedi pagine successive

3. Progettare un algoritmo che dato un vettore di numeri interi, restituisce il numero che appare più volte in tale vettore.

Soluzione Una possibile soluzione potrebbe prevedere l'utilizzo di una struttura dati dizionario in cui, durante una lettura sequenziale del vettore, vengono memorizzati i valori incontrati (da considerarsi come chiavi del dizionario) con associato il numero di occorrenze finora trovate per quel valore. Più precisamente, ogni volta che si incontra un valore k, si preleva (se presente) la coppia (e, k) dal dizionario e si inserisce la coppia $(e+1, k)$ (se k non è presente si inserisce $(1, k)$). Con una implementazione del dizionario tramite tabella hash, potremmo avere una buona complessità per quanto riguarda il tempo medio di esecuzione, ma un tempo $O(n^2)$ nel caso pessimo unitamente al costo in spazio richiesto per implementare la tabella hash. Per tali motivi, si presenta una soluzione che procede in loco, ordinando il vettore tramite algoritmo heapsort, e poi legge nuovamente il vettore ordinato alla ricerca della sequenza più lunga di valori contigui identici. Tale soluzione, nel caso pessimo, ha tempo di calcolo $T(n) = O(n \log n)$, con n lunghezza del vettore, in quanto il costo $O(n)$ per scorrere il vettore ordinato viene assorbito dal costo dell'esecuzione di heapsort.

In dettaglio, come soluzione si propone l'algoritmo `CercaOccorrenzeMassime` che utilizza le variabili `maxVal` e `maxOcc`, per memorizzare rispettivamente il numero finora incontrato con maggiori occorrenze contigue ed il relativo numero di occorrenze, e le variabili `current` e `cont`, per memorizzare rispettivamente il valore attualmente sotto controllo ed il relativo numero di occorrenze finora incontrate.

```
algoritmo CercaOccorrenzeMassime(v: Int[1..n]) --> Int

   //ordina il vettore in loco
   heapsort(v)

   //cerca il valore che appare piu' volte in posizioni contigue
   Int maxVal, current=v[1], maxOcc=0, cont=1
   for i = 1 .. n-1
      if (v[i+1]==current)
          cont++
      else
          if (cont>maxOcc)
              maxVal = v[i]
              maxOcc = cont
          endif
          current = v[i+1]
          cont = 1
      endif
   return maxVal
```

4. La rete stradale di una città è rappresentata tramite un grafo orientato pesato $G = (V, E, w)$. I vertici V rappresentano incroci, gli archi E rappresentano tratti di strada che collegano due incroci, ed il peso $w(u, v)$ di un arco (u, v) rappresenta il tempo in secondi per percorrere il tratto di strada dall'incrocio u all'incrocio v. In tale città vale la seguente regola: in ogni intervallo di 100 secondi, durante gli ultimi 5 secondi non si possono attraversare gli incroci. In altri termini, per ogni x intero, nell'intervallo di tempo $[x \times 100 + 95, (x+1) \times 100]$ non si possono attraversare incroci (per esempio, se si raggiunge un incrocio al secondo $x \times 100 + 97$ bisogna fermarsi per 3 secondi prima di procedere). Progettare un algoritmo che dato il grafo orientato pesato G, un incrocio di partenza p, ed un incrocio di arrivo a, restituisce il tempo minimo necessario per andare dal punto p al punto a.

Soluzione Essendo i pesi non negativi, si può procedere adottando l'algoritmo di Dijkstra. Bisogna però fare attenzione al calcolo delle distanze, in quanto se il resto della divisione per 100 di una distanza risulta essere maggiore o uguale a 95, bisognerà incrementarla della differenza fra 100 e tale distanza modulo 100. Questo è necessario per tenere in considerazione i ritardi derivanti dalla regole dei 100 secondi.

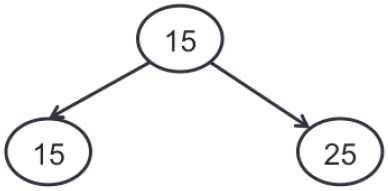
Cont.

```
algoritmo CamminoMinimo (Grafo G=(V,E,w), int p, int a) --> int

    // inizializzazione strutture dati
    int n = G.numNodi()
    int v, u, dist
    double D[1..n]
    for v = 1..n do
        D[v] = INFINITY
    endfor
    D[p] = 0
    CodaPriorita<int, double> Q; Q.insert(p, D[p]);

    // esecuzione algoritmo di Dijkstra
    while (not Q.isEmpty()) do
        u = Q.find()
        if (u == a)
            exit
        endif
        Q.deleteMin()
        for each v adiacente a u do
            // calcolo della nuova distanza di v usando il cammino minimo fino a u
            dist = D[u] + w(u,v)
            // controllo della regola dei 100 secondi su tale nuova distanza
            if (dist MOD 100 >= 95) then
                dist += 100 - (dist MOD 100)
            endif
            // aggiornamento coda con priorita'
            if (D[v] == INFINITY) then
                // prima volta che si incontra v
                D[v] = dist
                Q.insert(v, D[v])
            elseif (dist < D[v]) then
                // scoperta di un cammino migliore per raggiungere v
                Q.decreaseKey(v, D[v] - dist)
                D[v] = dist
            endif
        endfor
    endwhile

    // restituisce la distanza di a
    return(D[a])
```

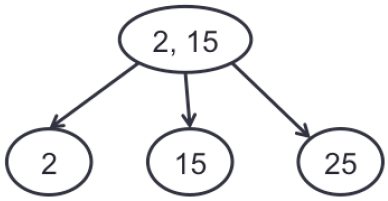
La complessità dell'algoritmo è la medesima dell'algoritmo di Dijkstra, ovvero $T(n, m) = O(m \log n)$, dove n è il numero di vertici ed m il numero di archi nel grafo.

Cont.

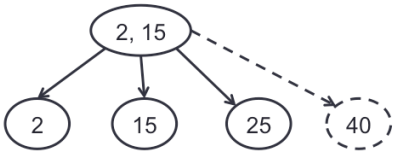
(a) INSERT(15). L'albero ha un unico nodo con valore 15.



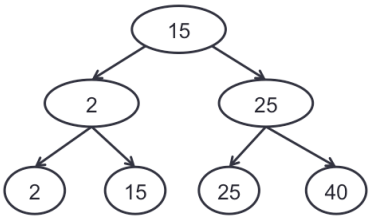
(b) INSERT(25). Viene aggiunta una foglia con valore 25 e creata una nuova radice, padre delle due chiavi. La radice contiene 15, valore più alto del sottoalbero sx.



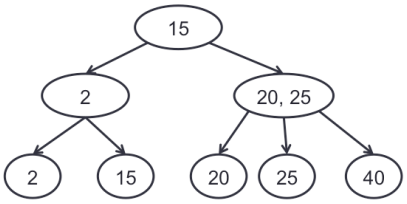
(c) INSERT(2). Il nodo 2 si aggiunge a sinistra essendo la chiave più piccola. La radice si aggiorna con i valori più alti nel sottoalbero sinistro e centrale. I vincoli degli alberi23 sono rispettati e non sono necessarie altre operazioni.



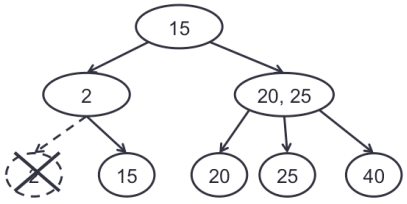
(d) INSERT(40). 40 occupa la posizione più a destra essendo la chiave più alta tra i figli della radice. Il vincolo sul numero dei figli non è rispettato ed è necessario uno split.



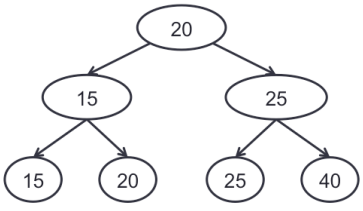
(e) Il risultato finale dello split, con aggiunta di un livello (radice).



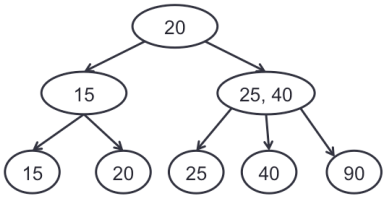
(f) INSERT(20). Il nodo 20 può essere aggiunto rispettando i vincoli sull'ordine delle chiavi e il numero di figlio. Si aggiorna il nodo intermedio (20, 25) visto che aumenta il numero di figli.



(g) DELETE(2). Il vincolo sul numero di figli non è rispettato dopo la cancellazione del nodo 2, per cui è necessaria una fusione (join).

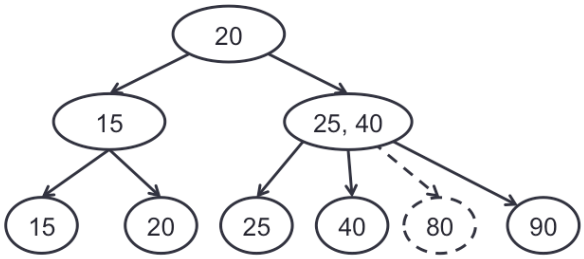


(h) Il risultato finale della fusione e propagazione fino alla radice. I nodi intermedi si aggiornano per tenere traccia dei valori più alti nei sottoalberi sx.

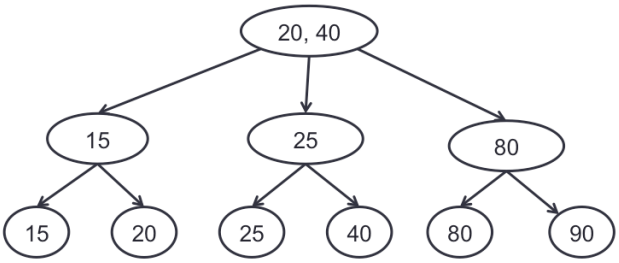


(i) INSERT(90). Il nodo 90 può essere aggiunto mantenendo l'ordine delle chiavi e rispettando il vincolo sul numero di figli.

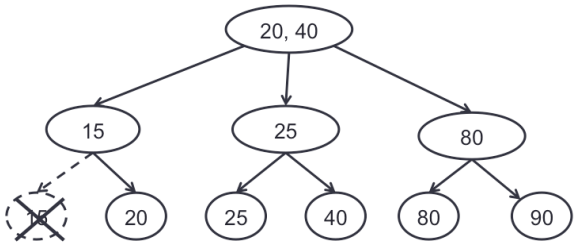
Cont.



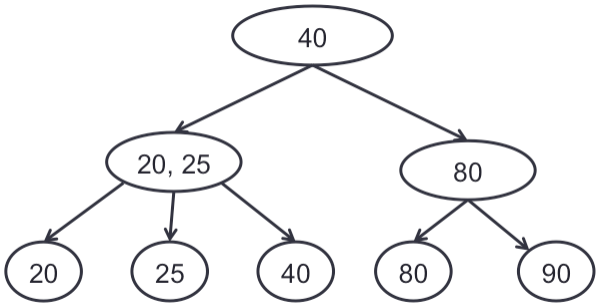
(j) INSERT(80). Il vincolo sul numero di figli non è rispettato dopo l'inserimento del valore 80, per cui è necessario uno split.



(k) Il risultato finale dello split e propagazione fino alla radice. I nodi intermedi e la radice si aggiornano per mantenere i valori più alti nei sottoalberi sinistro e centrale.



(l) DELETE(15). Il vincolo sul numero di figli non è rispettato dopo la cancellazione del nodo 2, per cui è necessaria una fusione (join).



(m) Il risultato finale della fusione e propagazione fino alla radice.

The End.